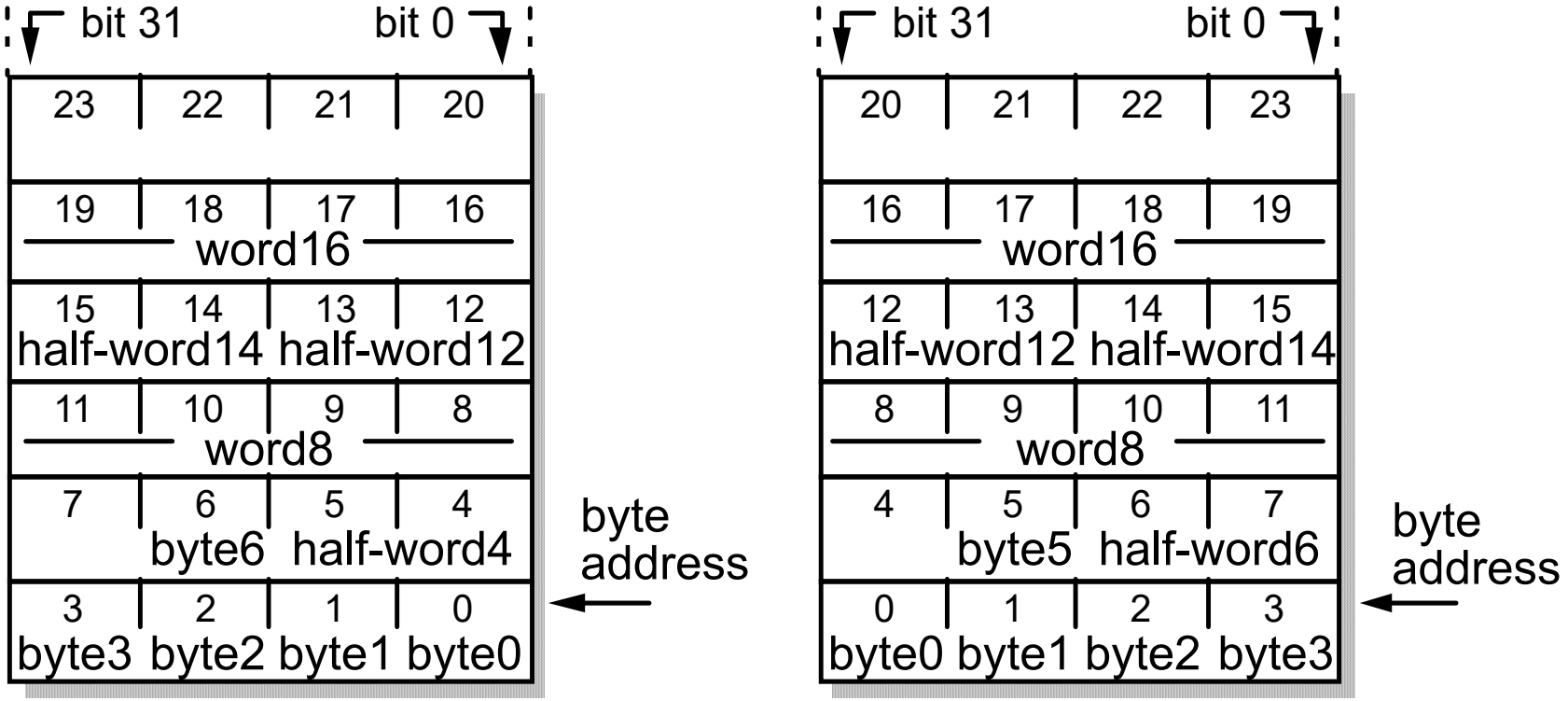
Chap. 5 The ARM Instruction Set

5.1 Introduction

- Data types
  - ARM processors supports six data types:
    - 8-bit signed and unsigned bytes
    - 16-bit signed and unsigned half-words; these are aligned on 2-byte boundaries
    - 32-bit signed and unsigned words; these are aligned on 4-byte boundaries
  - ARM instructions are all 32-bit words and must be word-aligned
  - Thumb instructions are half-words and must be aligned on 2-byte boundaries
  - All ARM operations are on 32-bit operands; the shorter data types are only supported by data transfer instructions

5.1 Introduction

- Memory organization
  - There are two ways to store words in a byte-addressed memory, depending on whether the least significant byte is stored at a lower or higher address than the next most significant byte
  - The issue causes significant practical difficulties when datasets are transferred between machines of opposite orderings
  - Most ARM chips remain strictly neutral in the dispute and can be configured to work with either memory arrangement, though they default to little-endian
  - This book we will assume a little-endian ordering



(a) Little-endian memory organization

(b) Big-endian memory organization

Fig. 5.1 Little- and big-endian memory organizations

5.1 Introduction

- Privileged modes
  - The current operating mode is defined by the bottom five bits of the CPSR
  - The relevant shaded registers shown in Fig. 2.1 replace the corresponding user registers and the current SPSR (Saved Program Status Register) also becomes accessible

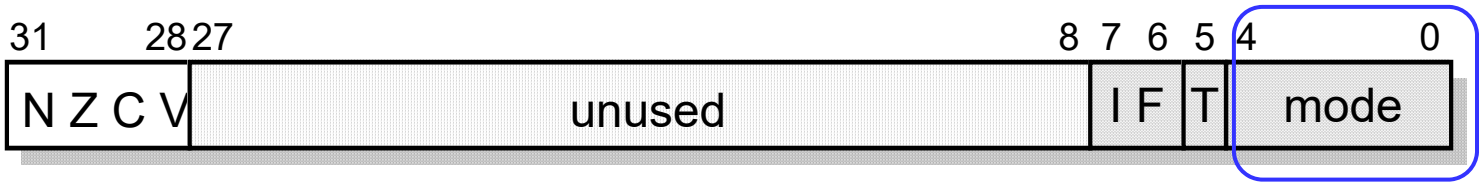


Fig. 2.2 ARM CPSR format

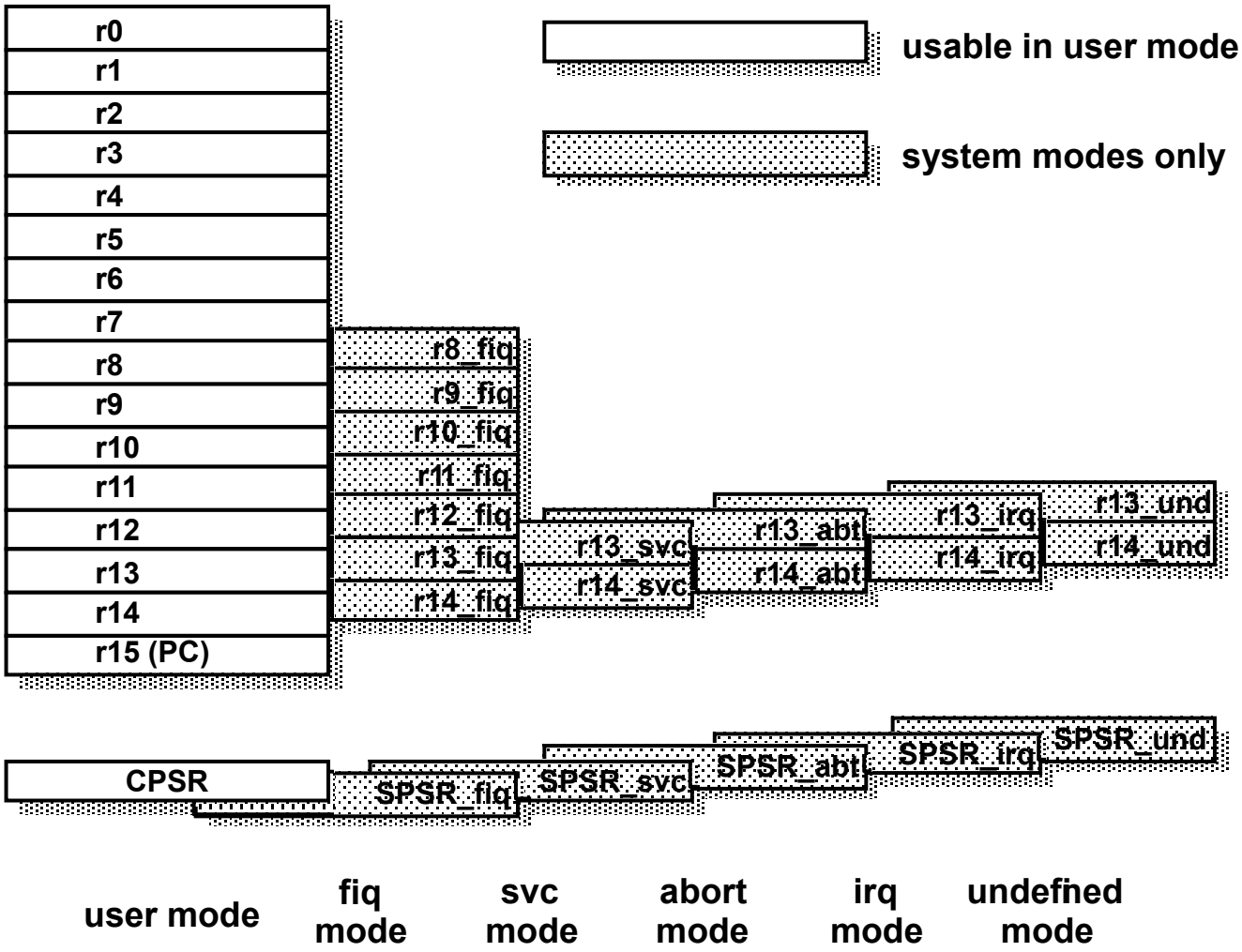


Fig. 2.1 ARM's visible registers

CPSR[4:0]	Mode	Use	Registers
10000	User	Normal user code	user
10001	FIQ	Processing fast interrupts	_fiq
10010	IRQ	Processing standard interrupts	_irq
10011	SVC	Processing software interrupts (SWIs)	_svc
10111	Abort	Processing memory faults	_abt
11011	Undef	Handling undefined instruction traps	_und
11111	System	Running privileged operating system tasks	user

Table 5.1 ARM operating modes and register usage

5.1 Introduction

- The SPSRs
  - This register is used to save the state of the CPSR (Current Program Status Register) when the
  - mode is entered in order that the user state can be fully restored when the user process is resumed
  - Often the SPSR may be untouched from the time the privileged mode is entered to the time it is used to restore the CPSR
  - If the privileged software is to be re-entrant then the SPSR must be copied into a general register and saved

5.2 Exceptions

- ARM exceptions
  - Exceptions generated as the direct effect of executing an instruction – Software interrupts, undefined instructions and prefetch aborts (instructions that are invalid due to a memory fault occurring during fetch)
  - Exceptions generated as a side-effect of an instruction – data aborts (a memory fault during a load or store data access)
  - Exceptions generated externally, unrelated to the instruction flow – Reset, IRQ and FIQ fall into this category

5.2 Exception Entry

- When an exception arises, ARM completes the current instruction as best it can and then departs from the current instruction sequence to handle the exception

- The processor performs the following sequence of actions:
  - It changes to the operating mode corresponding to the particular exception
  - It saves the address of the instruction following the exception entry instruction in r14 of the new mode
  - It saves the old value of the CPSR in the SPSR of the new mode
  - It disables IRQs by setting bit 7 of the CPSR
    - IRQs are disabled when any exception occurs.
    - FIQs are disabled when a FIQ occurs, and on reset
  - It forces the PC to begin executing at the relevant vector address (is normally a branch) in Table 5.2
    - normally the vector address will contain a branch to the relevant routine

- The two banked registers (r14 and r13) in each of the privileged modes are used to hold the return address and a stack pointer

Exception	Mode	Vector address
Reset	SVC	0x00000000
Undefined instruction	UND	0x00000004
Software interrupt (SWI)	SVC	0x00000008
Prefetch abort (instruction fetch memory fault)	Abort	0x0000000C
Data abort (data access memory fault)	Abort	0x00000010
IRQ (normal interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

Table 5.2 Exception vector addresses

Startup code

- Startup code for C/C++ program consists of the following actions
  - Disable all interrupt
  - Copy any initialized data from ROM to RAM
  - Zero the uninitialized data area
  - Allocate space for and initialize the stack
  - Initialize the processor's stack pointer
  - Create and initialize the heap
  - Execute the constructors and initializers for all global variables (C++ only)
  - Enable interrupts
  - Call main

5.2 Exceptions

- Exception return
  - Any modified user registers must be restored from the handler's stack
  - The CPSR must be restored from the appropriate SPSR
  - The PC must be changed back to the relevant instruction address in the user instruction stream
  - The last two steps cannot be carried out independently.
  - The case where the return address is in r14 (S: destination is pc):
    - To return from a SWI or undefined instruction trap use

      ```
      MOVS    pc,  r14
      ```

    - To return from an IRQ, FIQ or prefetch abort use

      ```
      SUBS    pc,  r14,  #4
      ```

    - To return from a data abort to retry the data access use

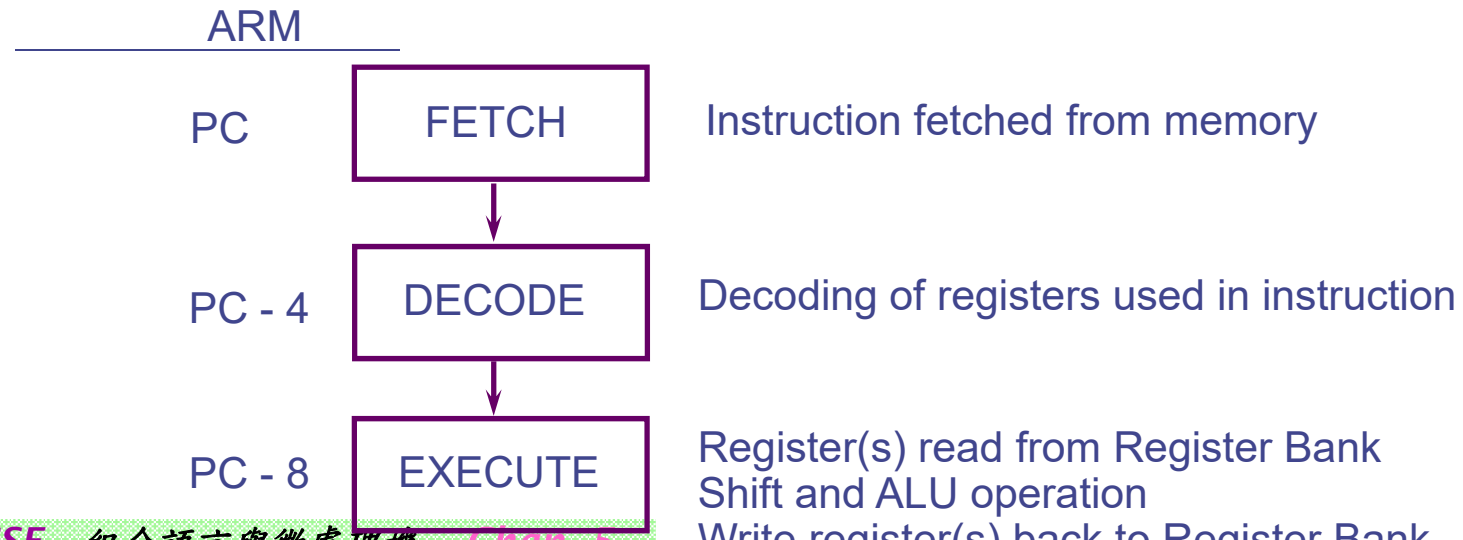
      ```
      SUBS    pc,  r14,  #8
      ```

  - The return address out onto a stack:
    - LDMFD r13!, {r0 – r3, pc}^ ; restore and return

CPSR is also restore

The Instruction Pipeline

Exception	*Address*	*Use*
Reset	-	lr is not defined
Data abort	lr-8	Points to the instructions that caused data abort
FIQ	lr-4	Return to next instruction
IRQ	lr-4	Return to next instruction
Prefetch Abort	lr-4	Points to the instruction that cause prefetch aboint
SWI	lr	Return to next instruction after SWI
Undefined Instruction	lr	Return to next instruction after undefined instruction



5.2 Exceptions

- Exception priorities
  - Since multiple exceptions can arise at the same time it is necessary to define a priority order to determine the order in which the exceptions are handled:
    - 1. Reset (highest priority)
    - 2. Data abort
    - 3. FIQ
    - 4. IRQ
    - 5. Prefetch abort
    - 6. SWI, undefined instruction

Some instruction execution example

- Example1
  - PRE
    - cpsr=nzcvqiFt_USER
    - r1 = 0x00000001
    - SUBS r1, r1, #1
  - POST
    - cpsr=nZCvqiFt_USER
    - r1 = 0x00000000
- Example2
  - PRE
    - cpsr=nzcvqiFt_USER
    - r0 = 0x00000000
    - r1 = 0x80000004
    - MOVS r0, r1, LSL #1
  - POST
    - cpsr=nzCvqiFt_USER
    - r0 = 0x00000008
    - r1 = 0x80000004
- Example3
  - PRE
    - cpsr=nzcVqift_USER
    - pc = 0x00008000
    - lr = 0x003fffff; lr=r14
    - r0 = 0x12
    - 0x00008000 SWI 0x123456
  - POST
    - cpsr=nzcVqift_SVC
    - spsr=nzcVqift_USER
    - pc = 0x00000008
    - lr = 0x00008004;
    - r0 = 0x12

5.3 Conditional execution

- Conditional execution
  - The condition field occupies the top four bits of the 32-bit instruction field:

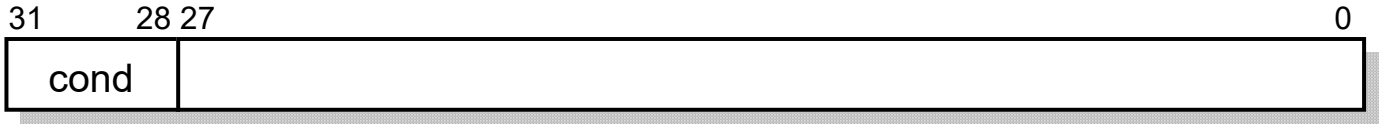


Fig. 5.2 The ARM condition code field

  - Each of the 16 values of the condition field causes the instruction to be executed or skipped according to the values of the N, Z, C and V flags in the CPSR
  - The conditions are given in Table 5.3
  - The 'always' condition (AL) may be omitted since it is the default condition

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

Table 5.3 ARM condition codes

Opcode [31:28]	Interpretation
0000	Equal / equals zero
0001	Not equal
0010	Carry set / unsigned higher or same
0011	Carry clear / unsigned lower
0100	Minus / negative
0101	Plus / positive or zero
0110	Overflow
0111	No overflow
1000	Unsigned higher
1001	Unsigned lower or same
1010	Signed greater than or equal
1011	Signed less than
1100	Signed greater than
1101	Signed less than or equal
1110	Always
1111	Never (do not use!)

Table 5.3 ARM condition codes

5.3 Conditional execution

- The ‘never’ condition
  - The ‘never’ condition (NV) should not be used
    - There are plenty of other ways to write no-ops
  - They may use this area of the instruction space for other purposes in the future
- Alternative mnemonics
  - This indicates that there is more than one way to interpret the condition field

```
CMP     r0, #5              ; if (r0 != 5) {
ADDNE   r1, r1, r0          ;   r1 := r1 + r0 - r2
SUBNE   r1, r1, r2          ; }
..
```

5.4 Branch and Branch with Link (B, BL)

- Binary encoding

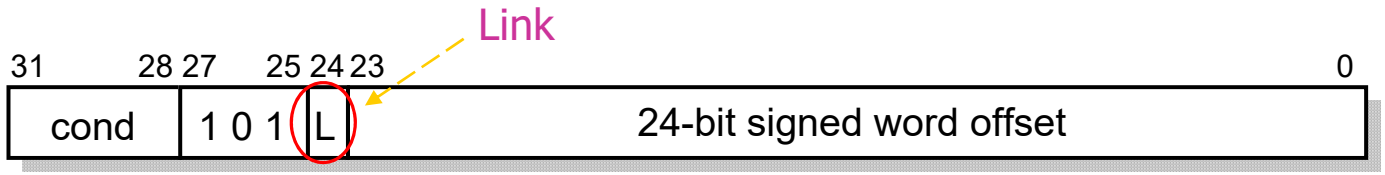


Fig. 5.3 Branch and Branch with Link binary encoding

- Description
  - Branch and branch with Link instructions cause the processor to begin executing instructions from an address computed by sign extending the 24-bit offset specified in the instruction
  - The Branch with Link variant, which has the L bit (bit 24) set, also moves the address of the instruction following the branch into the link register (r14) of the current processor mode

5.4 Branch and Branch with Link (B, BL)

- Assembler format :
  - B{L} {<cond>} <target address>
    - 'L' specifies the branch and link variant
    - '<cond>' should be one of the mnemonic extensions
    - '<target address>' is normally a label in the assembler code

- Example (conditional subroutine call)

```
..
CMP     r0, #5   ; if r0 < 5
BLLT    SUB1     ; then call SUB1
BLGE    SUB2     ; else call SUB2
..
```

5.5 Branch, Branch with Link and eXchange (BX, BLX)

- BX, BLX
  - These instructions are available on ARM chips which support the Thumb (16-bit) instruction set, and are a mechanism for switching the processor to execute Thumb instructions or for returning symmetrically to ARM and Thumb calling routines

- Binary encoding

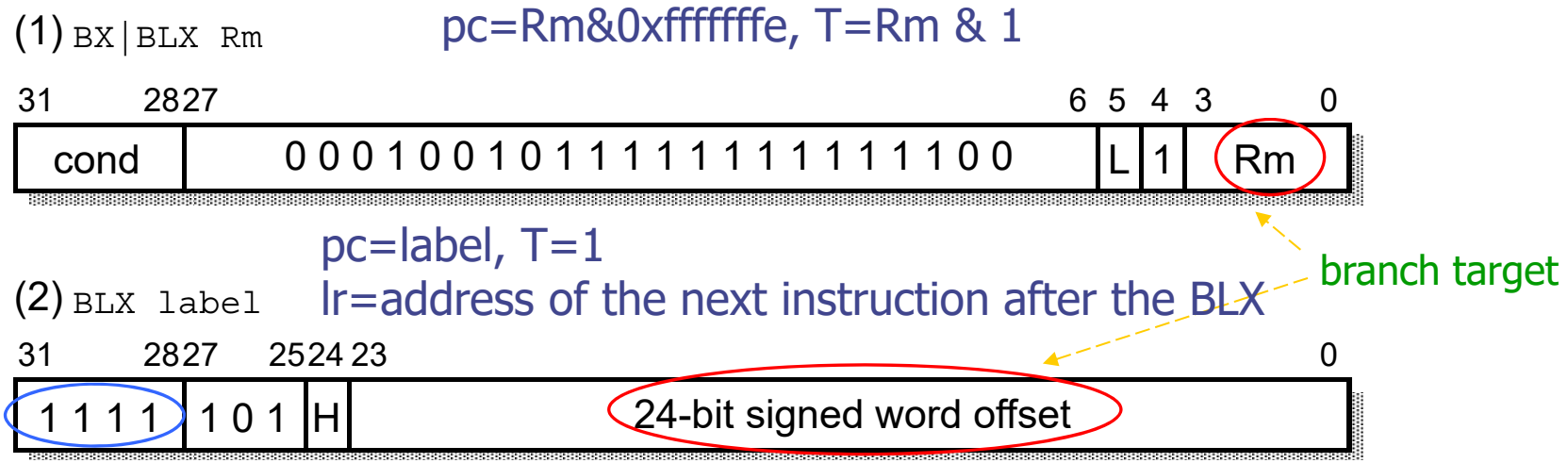
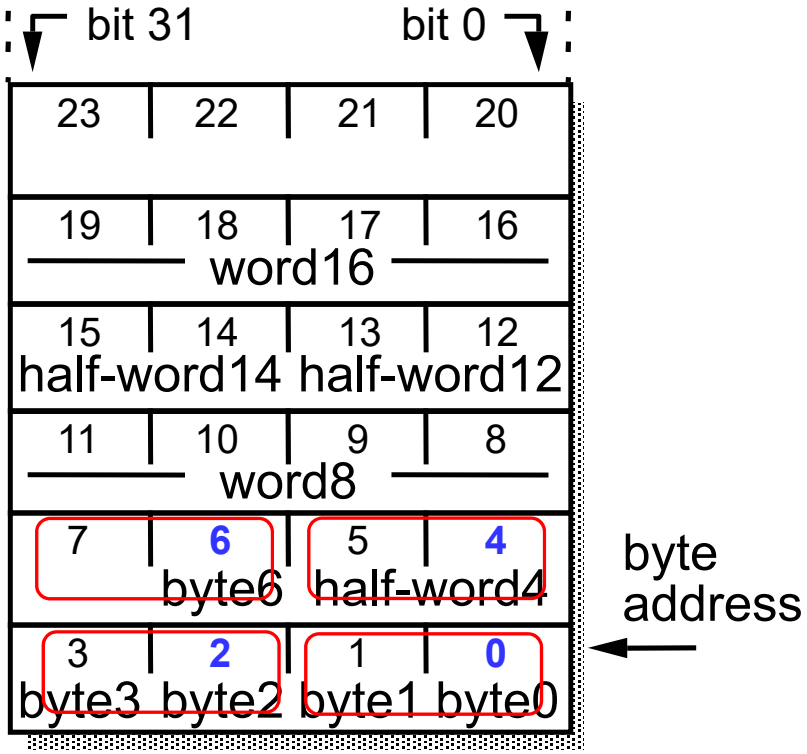


Fig. 5.4 Branch (with optional link) and exchange instruction binary encoding

5.5 Branch, Branch with Link and eXchange (BX, BLX)

- Description (Cont.)
  - In the *first format* the branch target is specified in a register, Rm
  - Bit[0] of Rm is copied into the **T** bit in the CPSR and bits[31:1] are moved into the PC:

    - If Rm[0] is 1, the processor switches to execute Thumb instructions and begins executing at the address in Rm aligned to a half-word boundary by clearing the bottom bit (…..0)
    - If Rm[0] is 0, the processor continues executing ARM instructions and begins executing at the address in Rm aligned to a word boundary by clearing Rm[1] (….00)
  - In the *second format* the branch target is an address computed by sign extending the 24-bit offset specified in the instruction
  - Format (1) instructions may be executed conditionally or unconditionally, but format (2) instructions are executed unconditionally



Little-endian memory organization

5.5 Branch, Branch with Link and eXchange (BX, BLX)

- Assembler format :

  1: B{L}X{<cond>} Rm

  2: BLX <target address>

  <target address> is normally a label

- Example

```
                CODE32                  ;  ARM code follows
  . .
                BLX         TSUB        ; call Thumb subroutine
          . .
                CODE16                   ; start of Thumb code
TSUB    . .                              ; Thumb subroutine
                BX          r14          ; return to ARM code
```

5.6 Software Interrupt (SWI)

- Software interrupt
  - The software interrupt instruction is used for calls to the operating system and is often called a 'supervisor call'

- Binary encoding

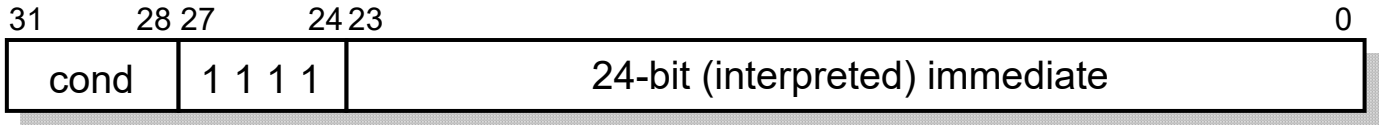


Fig. 5.5 Software interrupt binary encoding

5.6 Software Interrupt (SWI)

- Description
  - The processor actions are:
    1. Save the address of the instruction after the SWI in r14_svc
    2. Save the CPSR in SPSR_svc
    3. Enter supervisor mode and disable IRQs by setting CPSR[4:0] to 10011_2 and CPSR[7] to 1
    4. Set the PC to 08_{16} (is normally a branch to the SWI handler) and begin executing the instructions there
  - To return to the instruction after the SWI the system routine must not only copy r14_svc back into the PC, but it must also restore the CPSR from SPSR_svc

- Assembler format

  SWI{<cond>} <24-bit immediate>

5.6 Software Interrupt (SWI)

- Example

```
        ..
        BL      STROUT                  ; output following messag
        =       "Hello World",&0a,&0d,0
        ..                              ; return to here
        ..
STROUT  LDRB    r0, [r14], #1           ; get character
        CMP     r0, #0                  ; check for end marker
        SWINE   SWI_WriteC              ; if not end, print..
        BNE     STROUT                  ; .. and loop
        ADD     r14, #3                 ; align to next word
        BIC     r14, #3
        MOV     pc, r14                 ; return
```

5.6 Software Interrupt (SWI)

- Example

hexadecimal

```
        AREA    HelloW,CODE,READONLY ; declare code area
SWI_WriteC      EQU     &0           ; output character in r0
SWI_Exit        EQU     &11          ; finish program
        ENTRY                        ; code entry point
START   ADR     r1, TEXT             ; r1 -> "Hello World"
LOOP    LDRB    r0, [r1], #1         ; get the next byte
        CMP     r0, #0               ; check for text end
        SWINE   SWI_WriteC           ; if not end print ..
        BNE     LOOP                 ;   .. and loop back
        SWI     SWI_Exit             ; end of execution
TEXT    =       "Hello World",&0a,&0d,0
        END                          ; end of program source
```

SWI Exception

- The sample code of SWI handler
  - STMFD sp!, {r0-r12,r14}
  - LDR r10, [r14,#-4]
  - BIC r10, r10, #0xff000000
  - MOV r1,r13
  - MRS r2, spsr
  - STMFD r13!,{r2}
  - BL swi_jumptable
- The sample code of swi_jumptable
  - swi_jumptable
    - MOV r0, r10
    - B eventsSWIHandler

5.7 Data processing instructions

- ARM data processing instructions
  - are used to modify data values in registers
- Binary encoding: Fig. 5.6
- Description
  - When the instruction does not require all the available operands
    - MOV ignores Rn and CMP ignores Rd
    - The unused register field should be set to zero
  - Table 5.4
- We can write: ***MOV r1, -1*** because the assembler will transfer it to a suitable instruction to replace the original MOV instruction.

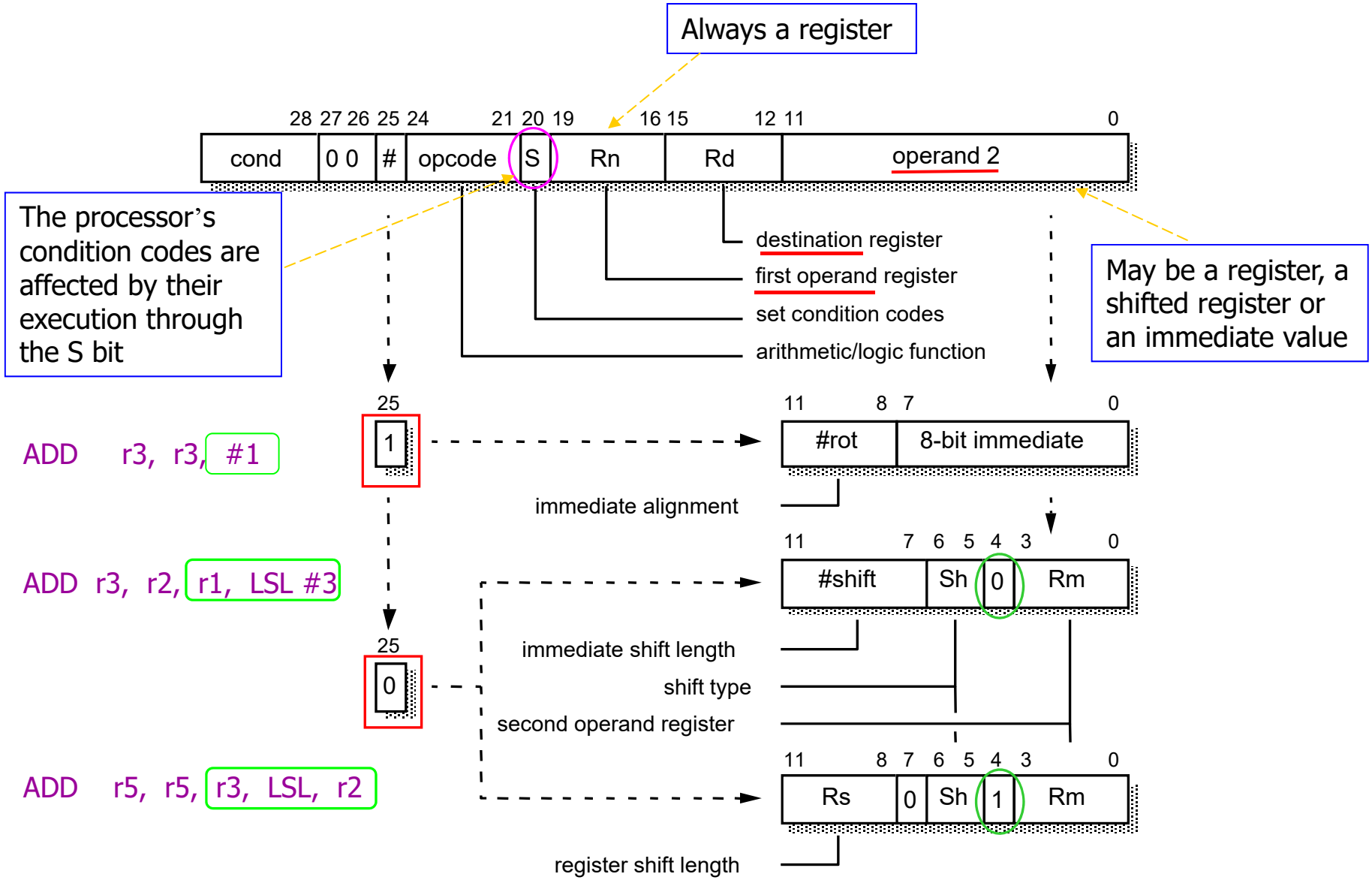
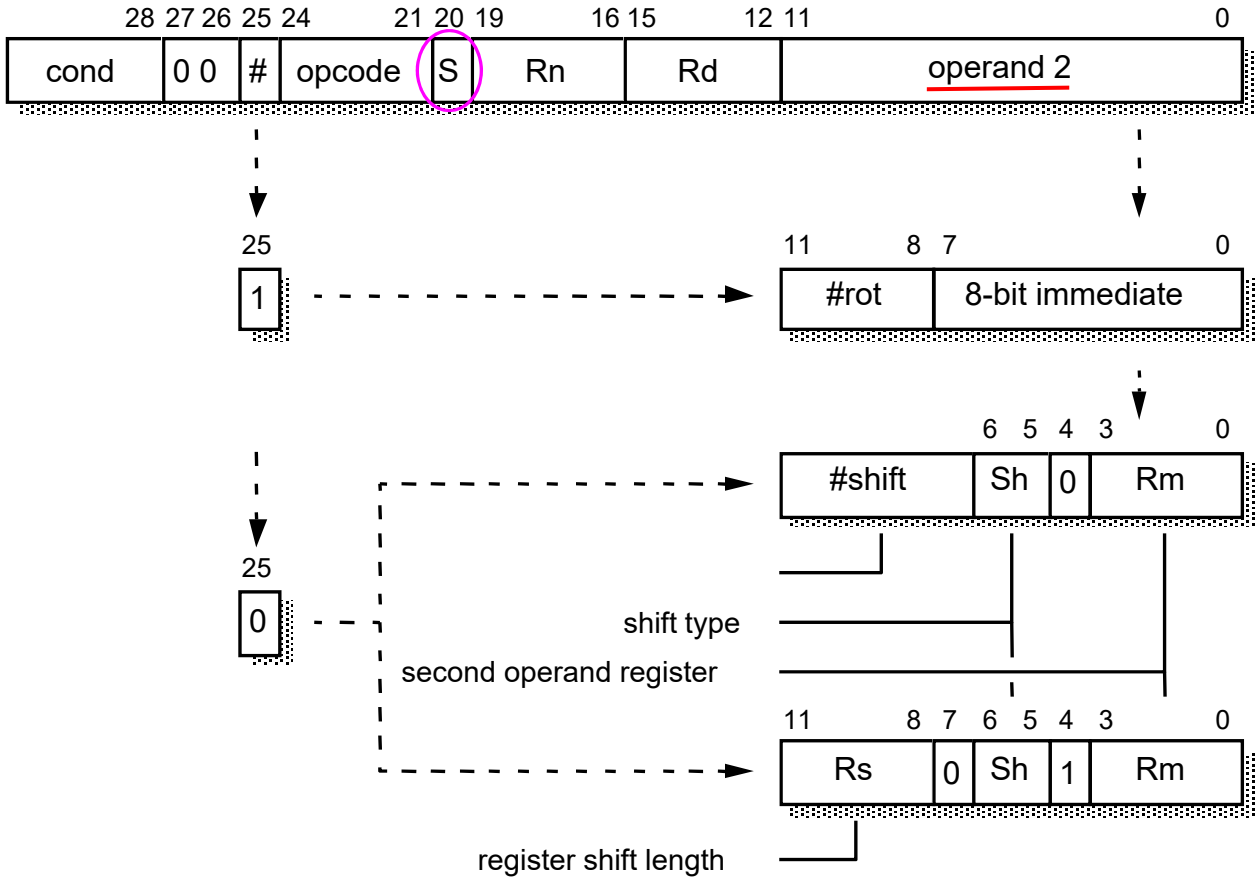


Fig. 5.6 Data processing instruction binary encoding


28 27 26 25 24 21 20 19 16 15 12 11 0
cond
0 0
#
opcode
S
Rn
Rd
operand 2
25
1
11 8 7 0
#rot
8-bit immediate
6 5 4 3 0
#shift
Sh
0
Rm
25
0
shift type
second operand register
11 8 7 6 5 4 3 0
Rs
0
Sh
1
Rm
register shift length

Opcode [24:21]	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	Rd := Rn AND Op2
0001	EOR	Logical bit-wise exclusive OR	Rd := Rn EOR Op2
0010	SUB	Subtract	Rd := Rn - Op2
0011	RSB	Reverse subtract	Rd := Op2 - Rn
0100	ADD	Add	Rd := Rn + Op2
0101	ADC	Add with carry	Rd := Rn + Op2 + C
0110	SBC	Subtract with carry	Rd := Rn - Op2 + C - 1
0111	RSC	Reverse subtract with carry	Rd := Op2 - Rn + C - 1
1000	TST	Test	Scc on Rn AND Op2
1001	TEQ	Test equivalence	Scc on Rn EOR Op2
1010	CMP	Compare	Scc on Rn - Op2
1011	CMN	Compare negated	Scc on Rn + Op2
1100	ORR	Logical bit-wise OR	Rd := Rn OR Op2
1101	MOV	Move	Rd := Op2
1110	BIC	Bit clear	Rd := Rn AND NOT Op2
1111	MVN	Move negated	Rd := NOT Op2

Table 5.4 ARM data processing instructions

5.7 Data processing instructions

- Assembler format
  - <op>{<cond>}{S} Rd, Rn, #<32-bit immediate>
  - <op>{<cond>}{S} Rd, Rn, Rm, {<shift>}
  - The S bit controls the effect of the instruction on the CPSR (N, Z, C and V)
- Examples
  - SUBS r2, r2, #1
    BEQ LABEL ; branch if r2 zero
    .....

5.8 Multiply instructions

- Multiply instructions
  - The result is a 64-bit product
  - Some forms of the instruction
    - Store the full result into two independently specified registers
    - Store only the least significant 32 bits into a single register
- Binary encoding

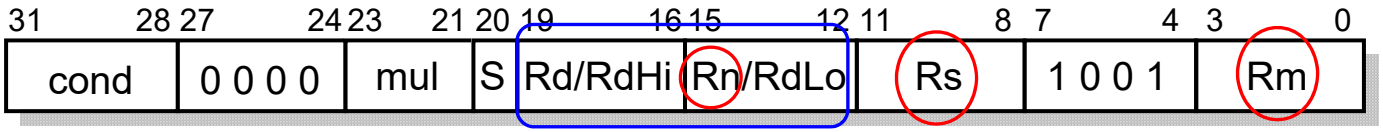


Fig. 5.7 Multiply instruction binary encoding

- Description (Table 5.5)

5.8 Multiply instructions

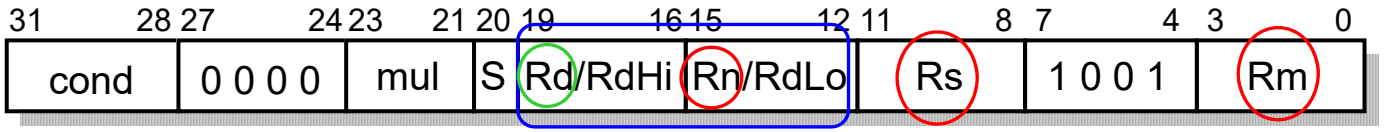
- Assembler formats
  - Instructions that produce the least significant 32 bits

    MUL{<cond>}{S} Rd, Rm, Rs

    MLA{<cond>}{S} Rd, Rm, Rs, Rn
  - Produce the full 64-bit result

    <mul>{<cond>}{S} RdHi, RdLo, Rm, Rs

- Notes
  - Rd, RdHi and RdLo should be distinct from Rm, and RdHi and RdLo should not be the same register
  - The 64-bit multipiles are available only on ARM7 versions with an 'M' in their name (ARM7DM, ARM7TM) and subsequent processors



Opcode [23:21]	Mnemonic	Meaning	Effect
000	MUL	Multiply (32-bit result)	Rd := (Rm * Rs) [31:0]
001	MLA	Multiply-accumulate (32-bit result)	Rd := (Rm * Rs + Rn) [31:0]
100	UMULL	Unsigned multiply long	RdHi:RdLo := Rm * Rs
101	UMLAL	Unsigned multiply-accumulate long	RdHi:RdLo += Rm * Rs
110	SMULL	Signed multiply long	RdHi:RdLo := Rm * Rs
111	SMLAL	Signed multiply-accumulate long	RdHi:RdLo += Rm * Rs

Table 5.5 Multiply instructions

is the 64 -bit number formed by concatenating RdHi and RdLo

Simple assignment is denoted

Accumulation

5.9 Count leading zeros (CLZ – architecture v5T only)

- Binary encoding

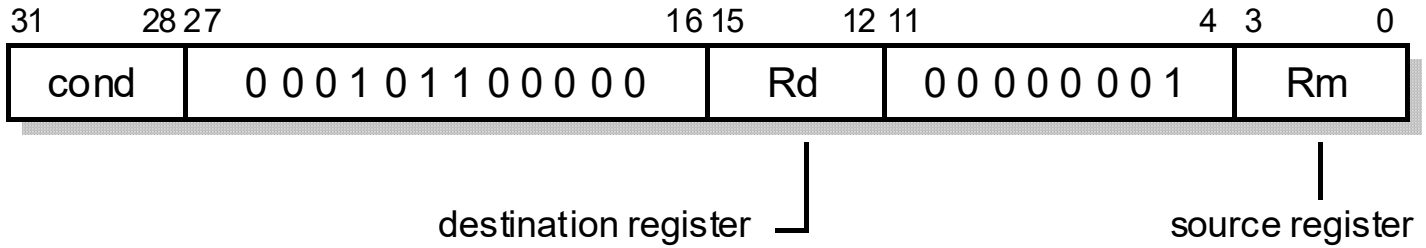


Fig. 5.8 Count leading zeros instruction binary encoding

- Description
  - The instruction sets Rd to the number of the bit position of the most significant 1 in Rm
  - If Rm is zero Rd will be set to 32
- Assembler format

CLZ{<cond>} Rd, Rm

5.9 Count leading zeros (CLZ – architecture v5T only)

- Example

```
MOV    r0,   #&100
CLZ    r1,   r0          ; r1:=23
```

- Notes
  - Only processors that implement ARM architecture v5T support the CLZ instruction

5.10 Single word and unsigned byte data transfer instructions

- Description (Fig. 5.9)
- Assembler format

```
LDRB   r0, [r1]      ; r0 := mem8 [r1]
```

  - Pre-indexed form

```
LDR|STR{<cond>}{B}  Rd, [Rn, <offset>]{!}
```

  - Post-indexed form

```
LDR|STR{<cond>}{B}{T}  Rd, [Rn], <offset>
```

  - PC-relative

```
LDR|STR{<cond>}{B}  Rd, LABEL
```

- Examples
  - Pre-indexed:

```
LDR   r0,  [ r1 , #4 ] !     ; r0 := mem32 [ r1 + 4 ]
                             ; r1 := r1 + 4
```

  - Post-indexed:

```
LDR   r0, [r1], #4           ; r0 := mem32 [r1]
                              ; r1 := r1 + 4
```

  - PC-relative:

```
            LDR    r1, UARTADD    ; range: 4 Kbytes
            ...
UARTADD     &      &1000000
```

5.10 Single word and unsigned byte data transfer instructions

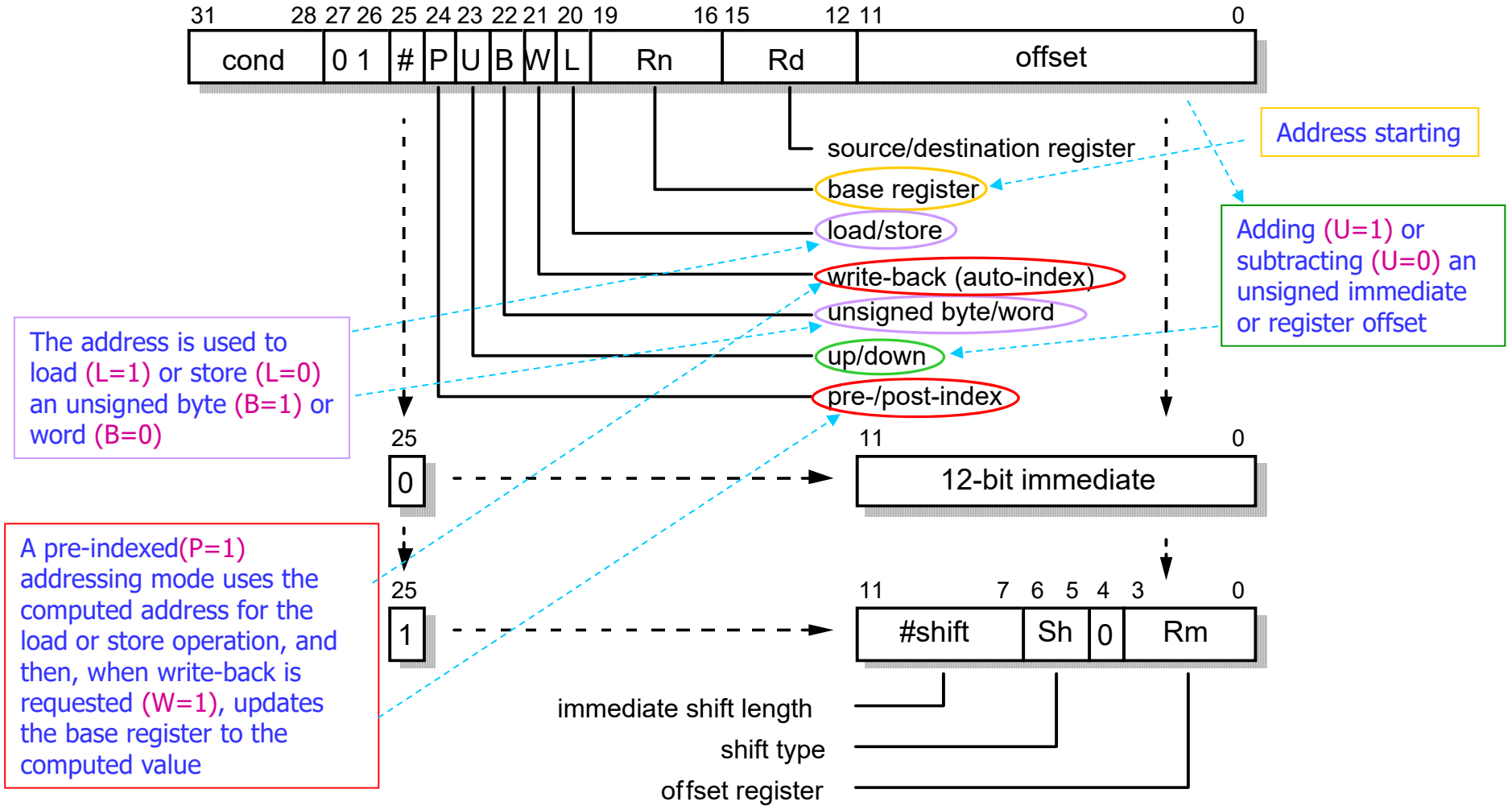
- Notes
  - PC should not be used as the offset register
  - Loading a byte into the PC should be avoided
  - Storing the PC to memory be avoided if possible
  - In general Rd, Rn and Rm should be distinct registers

```
LDR   r0, [r1]          ; r0 := mem32[r1]          P=0, W=0

LDR   r0, [ r1 , #4 ]   ; r0 := mem32[ r1 + 4 ]    P=1, W=0   pre-indexed

LDR   r0, [ r1 , #4 ] ! ; r0 := mem32[ r1 + 4 ]    P=1, W=1   pre-indexed
                        ; r1 := r1 + 4

LDR   r0, [r1], #4      ; r0 := mem32[r1]          P=0, W=1
                        ; r1 := r1 + 4
```



LDRLS pc, [r1, r0, LSL #2] Chap.3, p.35

Fig 5.9 Single word and unsigned byte data transfer instruction binary encoding

5.11 Half-word and signed byte data transfer instructions

- Binary encoding (Fig. 5.10)
- Description
  - The only relevant forms of this instruction format are
    - Load signed byte, signed half-word or unsigned half-word
    - Store half-word
    - An unsigned value is zero-extended to 32 bits when loaded
    - A signed value is extended to 32 bits by replicating the most significant bit of the data
- Assembler format

signed half-word

signed byte

  - The pre-indexed form

    LDR|STR{<cond>}H|SH|SB Rd, [Rn, <offset>]{!}

  - The post-indexed form

    LDR|STR{<cond>}H|SH|SB Rd, [Rn], <offset>

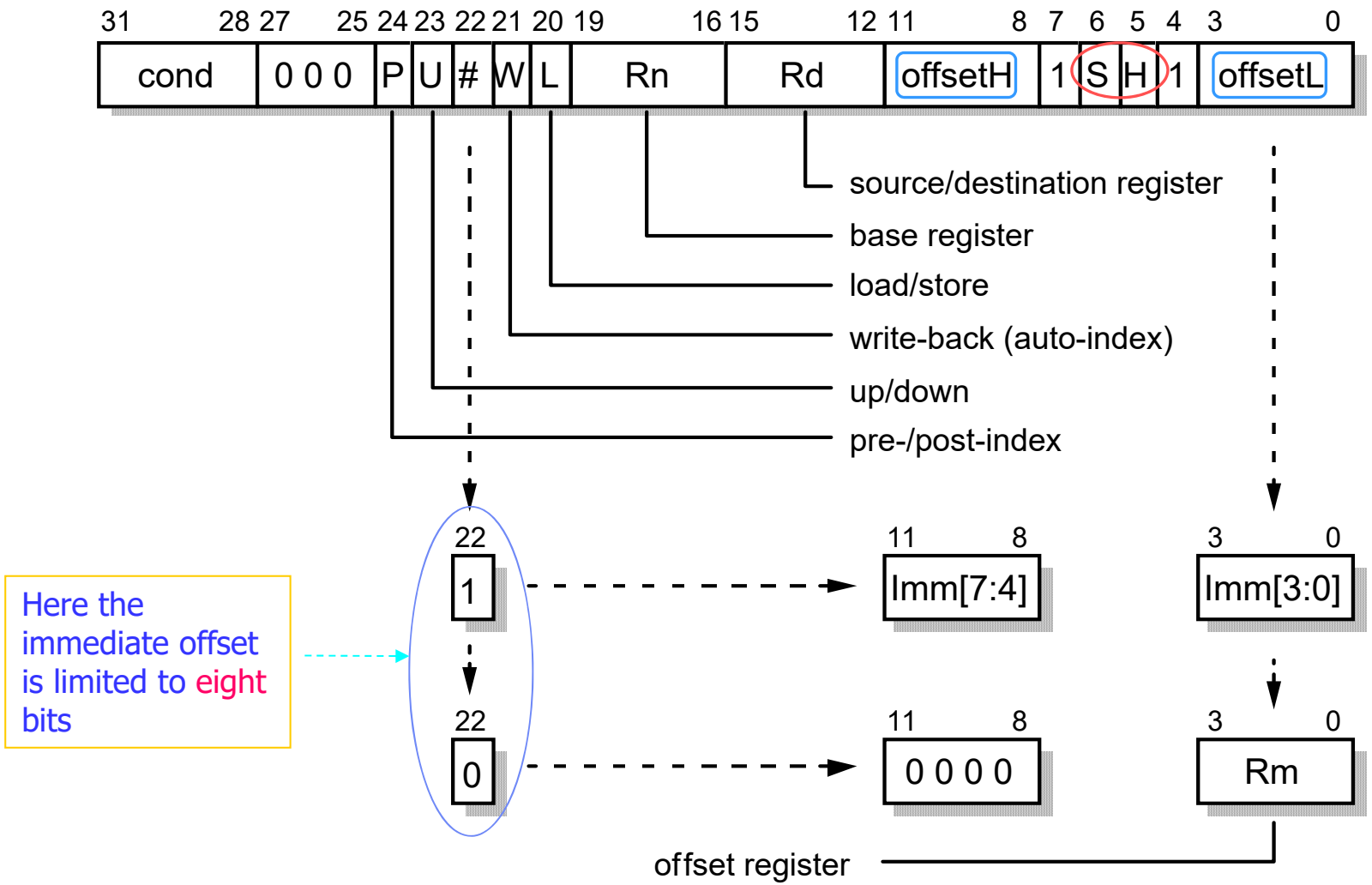


Fig. 5.10 Half-word and signed byte data transfer instruction binary encoding

5.11 Half-word and signed byte data transfer instructions

- Notes
  - All half-word transfers should use half-word aligned address

S	H	Data type
1	0	Signed byte
0	1	Unsigned half-word
1	1	Signed half-word

Table 5.6 Data type encoding

Example

- To expand an array of signed half-words into an array of words

```
        ADR     r1, ARRAY1
        ADR     r2, ARRAY2
        ADR     r3, ENDARR1
LOOP    LDRSH   r0, [r1], #2
        STR     r0, [r1], #4
        CMP     r1, r3
        BLT     LOOP
```

5.12 Multiple register transfer instructions

- Binary encoding (Fig. 5.11)

- Description
  - The register list in the bottom 16 bits of the instruction includes a bit for each visible register, with bit 0 controlling whether or not r0 is transferred….

- Assembler format

```
LDMIA     r0!,   {r2-r9}     ;  IA: increment after   W=1
STMFD     r13!,  {r2-r9}     ;  FD: full descending

LDMIA     r1,  {r0, r2, r5}  ; W=0
```

  - Normal form

    LDM|STM{<cond>}<add mode> Rn{!}, <registers>

  - In a non-user mode, the CPSR may be restored

    LDM{<cond>}<add mode> Rn{!}, <registers + PC>^

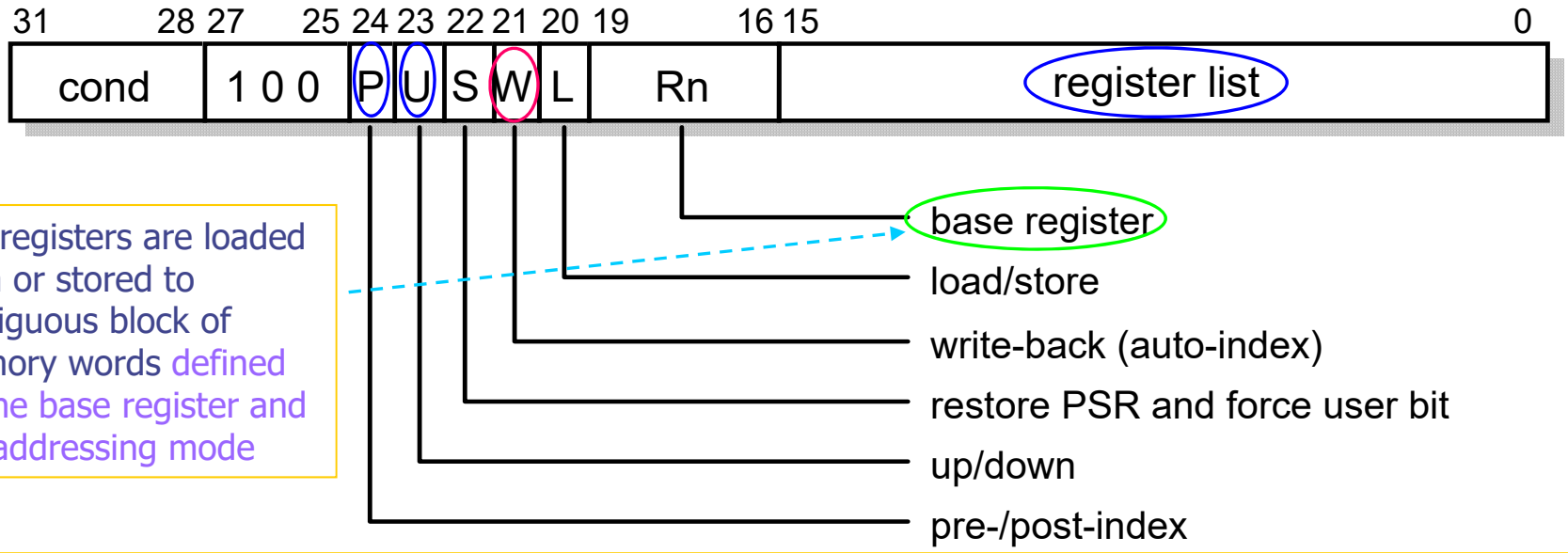
  - In a non-user mode, the user registers may be saved or restored

    LDM|STM{<cond>}<add mode> Rn, <registers - PC>^

Incremented (**U**=1) or decremented (**U**=0) Before (**P**=1) or after (**P**=0) each transfer

```
LDMIA    r0!,   {r2-r9}    ; IA: increment after   U=1, P=0,  W=1
STMFD    r13!,  {r2-r9}    ; FD: full descending
                             (decrement before)   U=0, P=1,  W=1

LDMIA    r1,  {r0, r2, r5}   ; W=0
```



The registers are loaded from or stored to contiguous block of memory words defined by the base register and the addressing mode

Special forms:

If the PC is in the register list of a load multiple and the **S** bit is set, the SPSR of the current mode will be copied into the CPSR, giving an atomic return and restore state instruction.

```
SUB1    STMFD   r13!, {r0-r2,r14}; save work regs & link
        BL      SUB2
        ..
        LDMFD   r13!, {r0-r2,pc} ; restore work regs & return
```

P.12

5.12 Multiple register transfer instructions

- Notes
  - Specifying the PC in an STM should be avoided
  - The base register may be specified in the transfer list, but write-back should not be specified in the same instruction

		Ascending		Descending	
		Full	Empty	Full	Empty
Increment	Before	STMIB STMFA			LDMIB LDMED
	After		STMIA STMEA	LDMIA LDMFD	
Decrement	Before		LDMDB LDMEA	STMDB STMFD	
	After	LDMDA LDMFA			STMDA STMED

Table 3.1

5.13 Swap memory and register instructions(SWP)

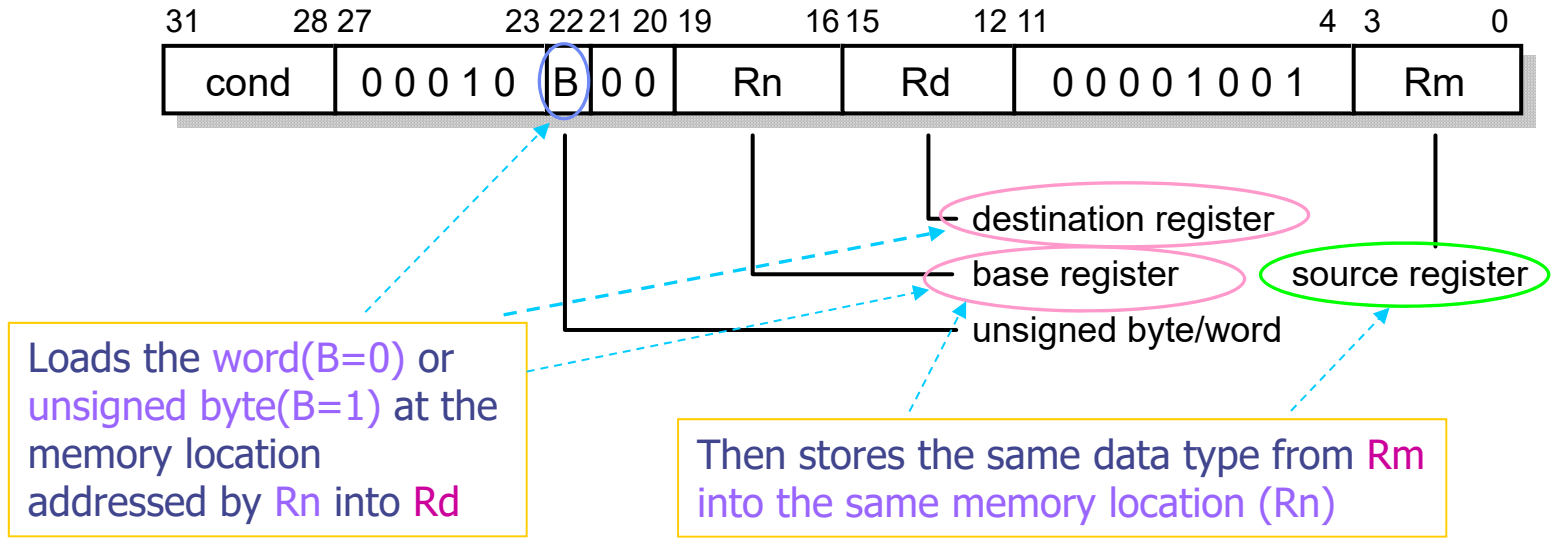
- Swap instructions
  - combine a load and a store of a word or an unsigned byte in a single instruction

Mem[Rn]=Rm, Rd=Mem[Rn]

SWP{<cond>}{B} Rd, Rm, [Rn]

SWPB r1, r1, [r0] ; exchange byte

- Binary encoding



Rd and Rm may be the same register

5.13 Swap memory and register instructions(SWP)

- Assembler format

  SWP{<cond>}{B} Rd, Rm, [Rn]

- Example

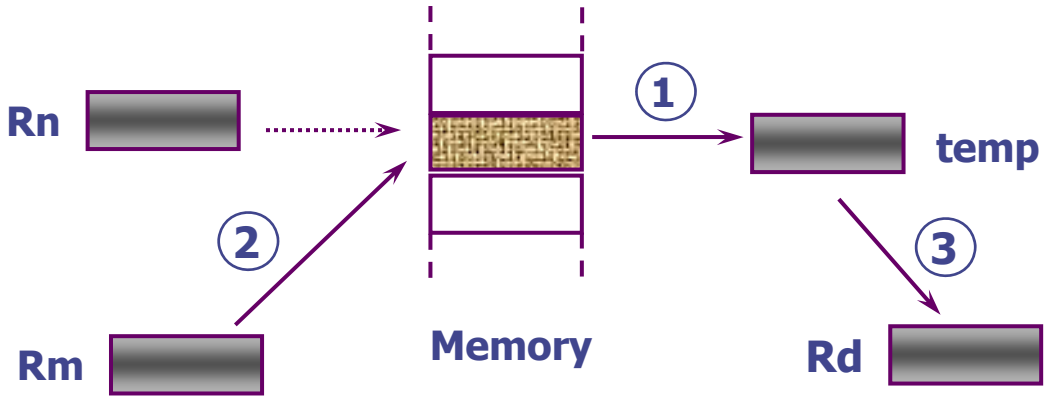
```
ADR     r0, SEMAPHORE
SWPB    r1, r1, [r0]          ; exchange byte
```

Notes

- The PC should not be used as any of the registers in this instruction
- The base register (Rn) should not be the same as either the source (Rm) or the destination (Rd) register

Swap and Swap Byte Instructions

- Atomic operation of a memory read followed by a memory write which moves byte or word quantities between registers and memory.
- Syntax:
  - SWP {<cond>} {B} Rd, Rm, [Rn]



- Thus to implement an actual swap of contents make Rd = Rm.
- The compiler cannot produce this instruction.

Ref. [8]

SWAP instruction

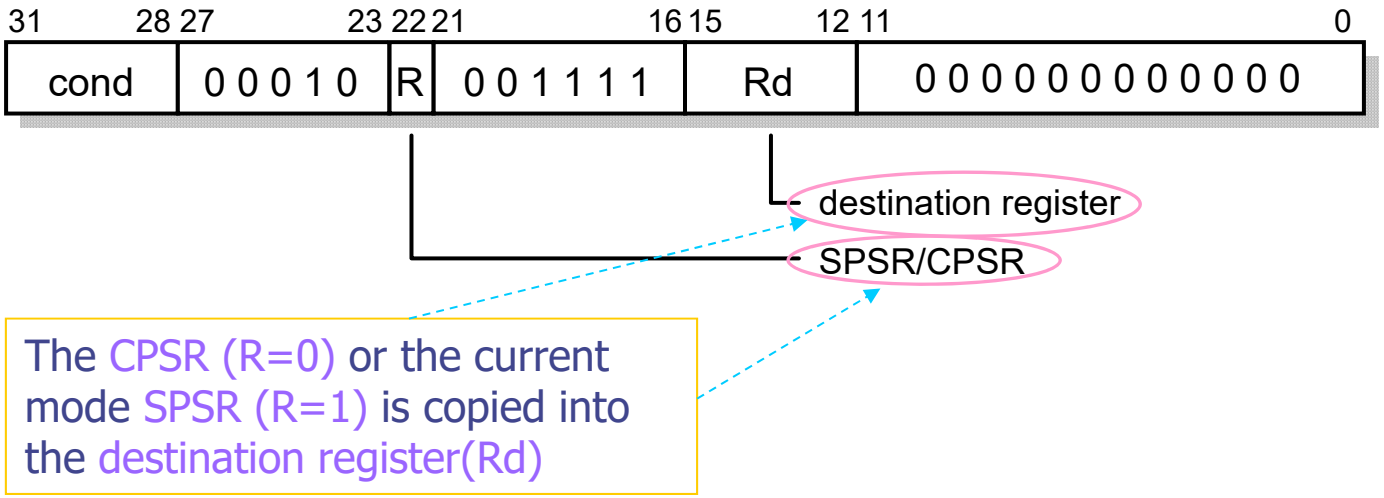
- The swap instruction is a special case of a load-store instructions. This instruction is an atomic operation.
- PRE
  - mem32[0x9000] = 0x12345678
  - r0 = 0x00000000
  - r1 = 0x11112222
  - r2 = 0x00009000
- SWP r0, r1, [r2]
- POST
  - mem32[0x9000] = 0x11112222
  - r0 = 0x12345678
  - r1 = 0x11112222
  - r2 = 0x00009000

Example of SWAP

- Semaphore:
  - Provide a mutual exclusive access control to the shared resource.
- spin
  - MOV r1, = semaphore
  - MOV r2, #1
  - SWP r3, r2, [r1]
  - CMP r3, #1
  - BEQ spin

5.14 Status register to general register transfer instructions

- When it is necessary to save or modify the contents of the CPSR or the SPSR of the current mode
  - Those contents must first be transferred into a general register
  - The selected bits modified
  - The value returned to the status register
- Binary encoding



5.14 Status register to general register transfer instructions

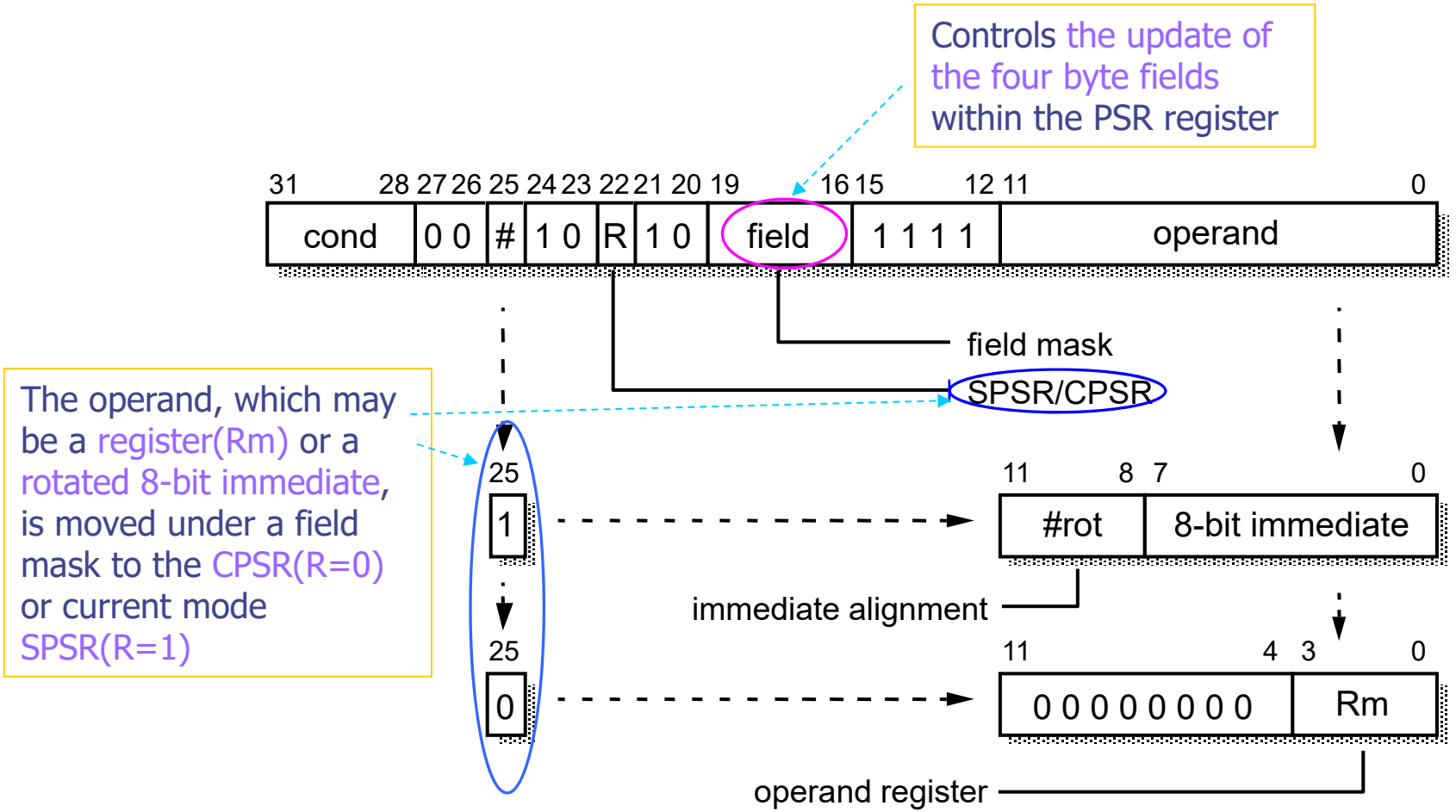
- Assembler format

  MRS{<cond>} Rd, CPSR|SPSR

- Notes
  - The SPSR form should not be used in user or system mode since there is no accessible SPSR in those modes

5.15 General register to status register transfer instructions

- Binary encoding



5.15 General register to status register transfer instructions

- Assembler format

  MSR{<cond>} CPSR_f|SPSR_f, #<32-bit immediate>

  MSR{<cond>} CPSR_<field>|SPSR_<field>, Rm

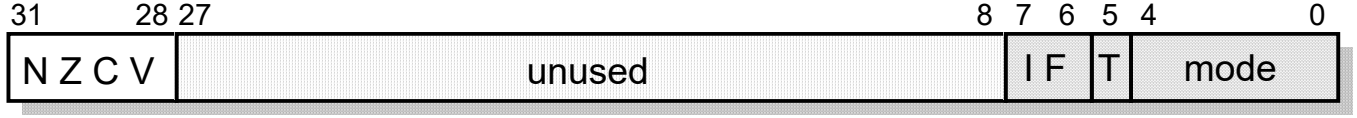
  <field> is one of:

  - c PSR[7:0]
  - x PSR[15:8]
  - s PSR[23:16]
  - f PSR[31:24]

- Examples

```
MRS     r0, CPSR                ; move the CPSR to r0
ORR     r0, r0, #&20000000      ; set bit 29 of r0
MSR     CPSR_f, r0              ; move back to CPSR
```

- Notes
  - When an immediate operand is used, only PSR[31:24] may be selected.
  - Attempts to modify any of CPSR[23:0] whilst in user mode have no effect



Program status register instructions

- The following example shows how to enable IRQ interrupts by clearing the I masks in the c field of CPSR.

- PRE
  - cpsr = nzcvqIFt_SVC
  - MRS r1, cpsr
  - BIC r1, r1, #0x80
  - MSR cpsr_c, r1

- POST
  - cpsr = nzcvqiFt_SVC

Why can user-level code not disable interrupts

- The following code illustrates how a malicious user destroies all the currently active programs.

```
        MSR    CPSR_f, #&c0
HERE    B      HERE
```

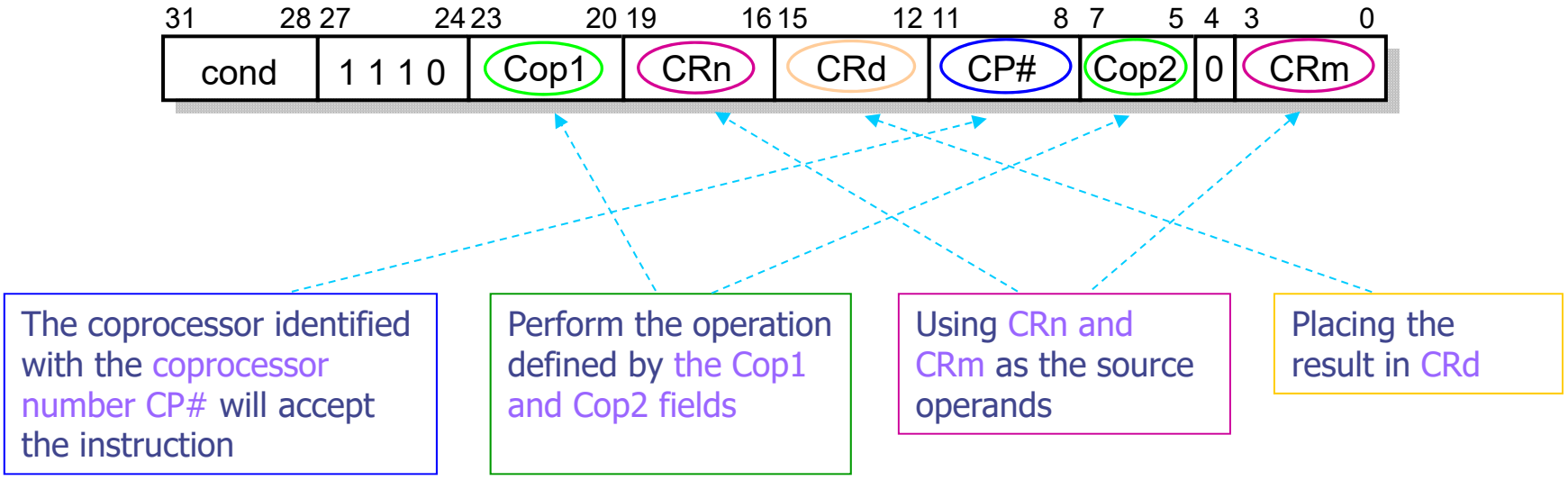
- The OS cannot regain control.
- The OS usually establishes a regular periodic interrupt from a hardware timer.

5.16 Coprocessor instructions

- Coprocessor instructions
  - The most common use of a coprocessor is the system coprocessor used to control on-chip functions such as the cache and memory management unit on the ARM720
- Coprocessor registers
  - ARM coprocessors have their own private register sets and their state is controlled by instructions that mirror the instructions that control ARM registers
- Coprocessor data operations
  - Coprocessor data operations are completely internal to the coprocessor and cause a stage change in the coprocessor registers
- Coprocessor data transfers
  - The ARM generates the memory address, but the coprocessor controls the number of word transferred
  - A coprocessor may perform some type conversion as part of the transfer
- Coprocessor register transfers
  - Move values between ARM and coprocessor registers

5.17 Coprocessor data operations

- Coprocessor data operations
  - The standard format follows the 3-address form of ARM's integer data processing instructions, but other interpretations of all the coprocessor fields are possible.
  - Undefined instruction exception is raised if the instruction is not accepted.
- Binary encoding



5.17 Coprocessor data operations

- Assembler format

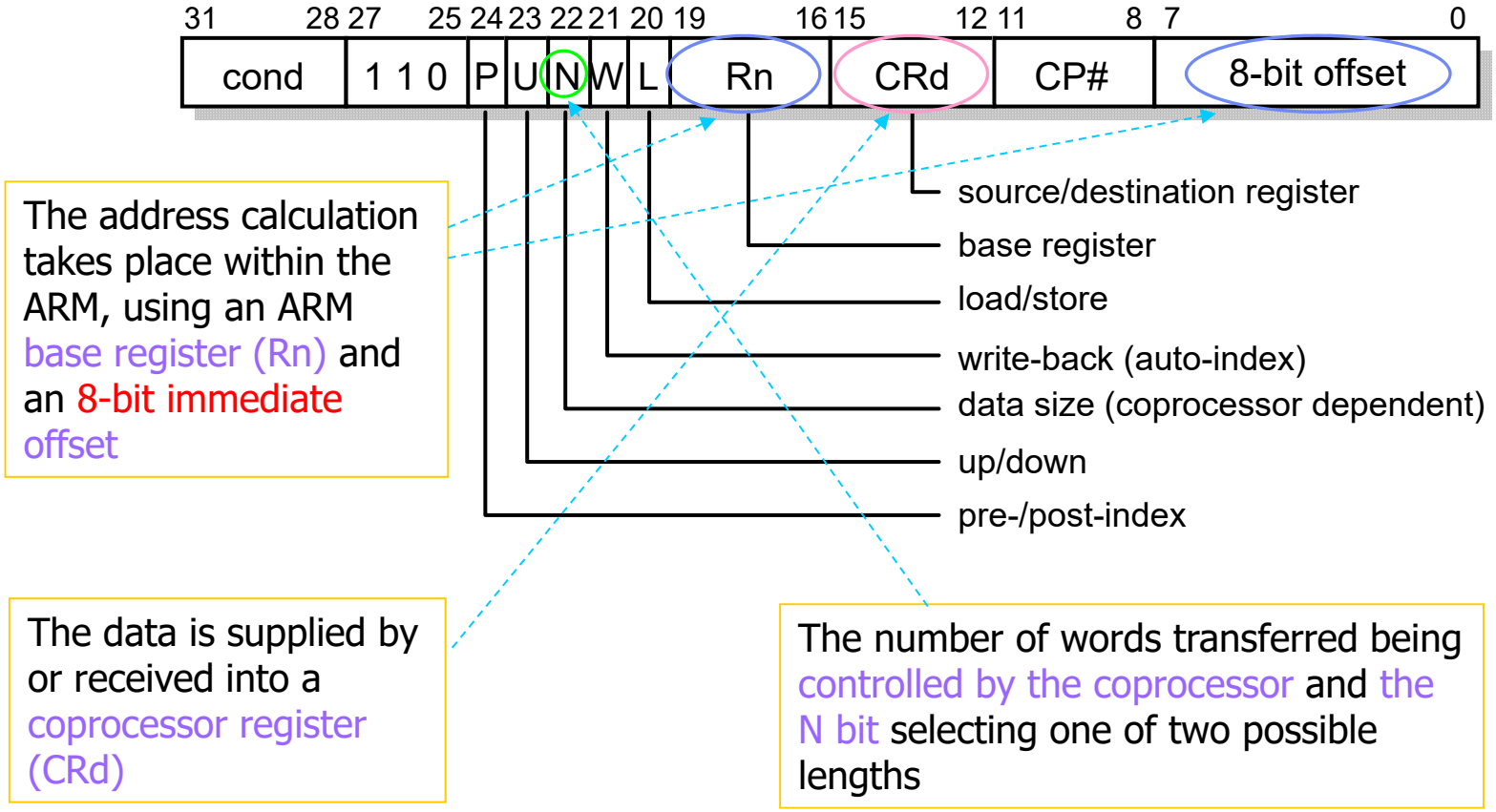
  CDP{<cond>} <CP#>, <Cop1>, CRd, CRn, CRm{, <Cop2>}

- Notes
  - The interpretation of the Cop1, CRn, CRd, Cop2 and CRm fields is coprocessor-dependent. It provides great flexibility.

- Examples
  - CDP p2, 3, C0, C1, C2
  - CDPEQ p3, 6, C1, C5, C7, 4

5.18 Coprocessor data transfers

- Binary encoding



5.18 Coprocessor data transfers

- Assembler format
  - The pre-indexed form

    LDC|STC{<cond>}{L} <#CP>, CRd, [Rn, <offset>]{!}
  - The post-indexed form

    LDC|STC{<cond>}{L} <#CP>, CRd, [Rn], <offset>
- Examples
  - LDC p6, C0, [r1]
  - STCEQL p5, C1, [r0], #4
- Notes
  - If the address is not word-aligned the two least significant bits will be ignored
  - During the data transfer the ARM will not respond to interrupt requests, so coprocessor designers should be careful not to compromise the system interrupt response time by allowing very long data transfer
  - If no coprocessor accepts the instruction, the undefined instruction traps.

5.19 Coprocessor register transfers

- Coprocessor register transfers
  - Allow an integer generated in a coprocessor to be transferred directly into a ARM register or the ARM condition code flags
    - A floating-point FIX operation which returns the integer to an ARM register
    - A floating-point comparison which returns the result of the comparison directly to the ARM condition code flags
    - A FLOAT operation which takes an integer value from an ARM register and sends it to the processor
- Binary encoding (Fig. 5.17)

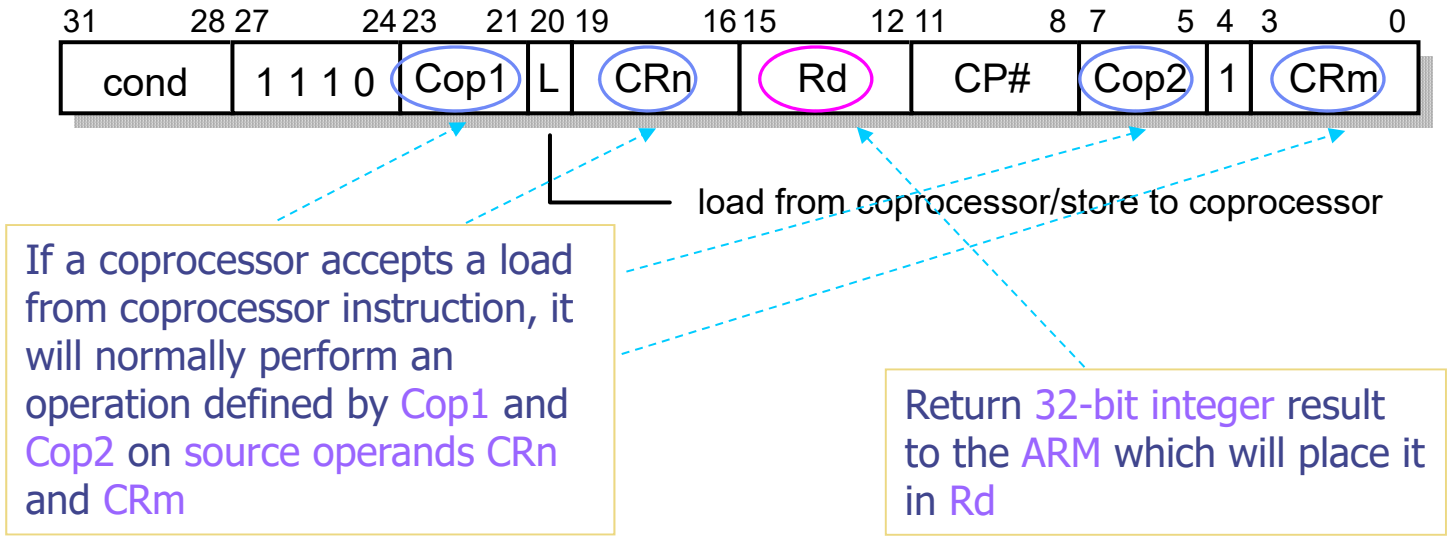


Fig. 5.17 Coprocessor register transfer instruction binary encoding

5.19 Coprocessor register transfers

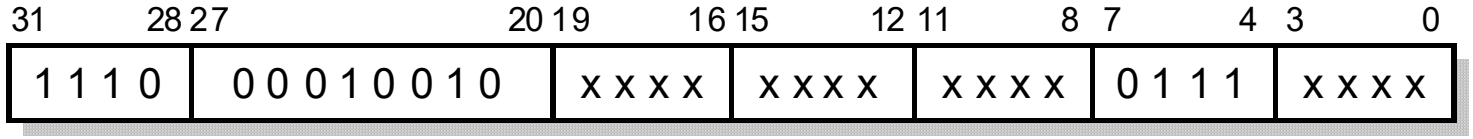
- Assembler format
  - Move to ARM register from coprocessor

    MRC{<cond>} <CP#>, <Cop1>, Rd, CRn, CRm{, <Cop2>}
  - Move to coprocessor from ARM register

    MCR{<cond>} <CP#>, <Cop1>, Rd, CRn, CRm{, <Cop2>}
- Examples
  - MCR p14, 3, r0, C1, C2
  - MRCCS p2, 4, r3, C3, C4, 6
- Notes
  - The coprocessor must perform some internal work to prepare a 32-bit value for transfer to the ARM, it will often be necessary for the coprocessor handshake to 'busy-wait' while the data is prepared
  - Transfers from the ARM to the coprocessor are generally simpler since any data conversion work can take place in the coprocessor after the transfer has completed

5.20 Breakpoint instruction (BKPT)

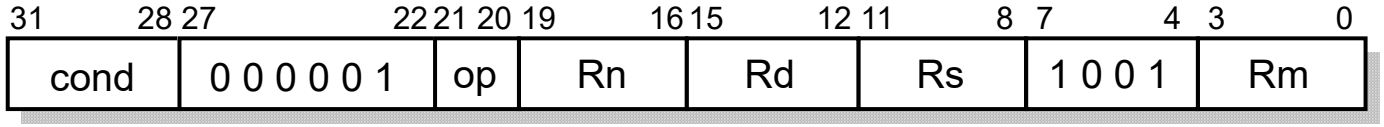
- Breakpoint instruction (BKPT)
  - Breakpoint instructions are used for software debugging purposes
  - This instruction causes the processor to take a prefetch abort when the debug hardware unit is configured appropriately
- Binary description



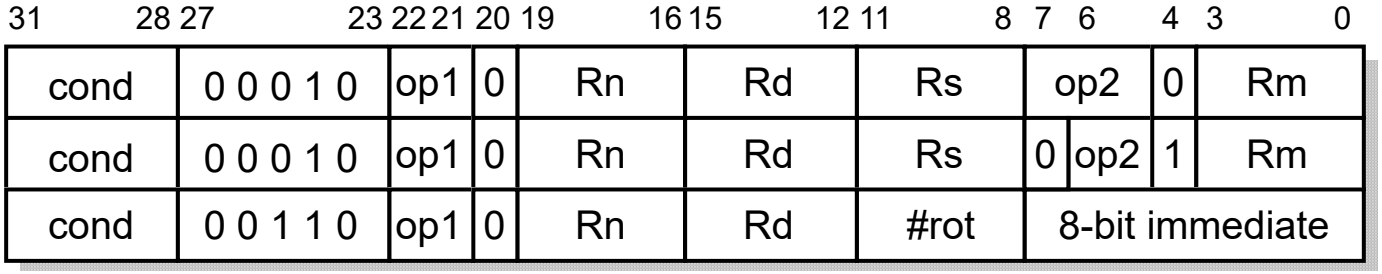
- Notes
  - Only processors that implement ARM architecture v5T support the BRK instruction
  - BRK instruction are unconditional

5.21 Unused instruction space

- Unused instruction space
  - not all of the 2^{32} instruction bit encodings have been assigned meanings
  - are available for future instruction set extensions
- Unused arithmetic instructions - This would be a likely encoding, for example, for an integer divide instruction

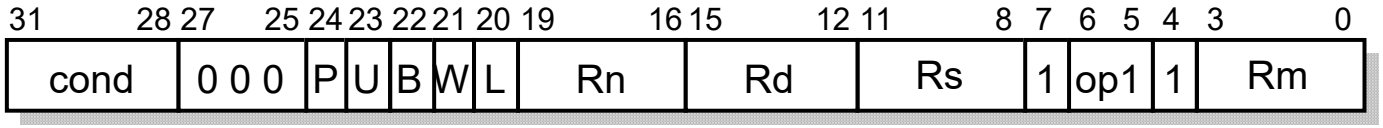


- Unused control instructions - The gaps here could be used to encode other instructions that affect the processor operating mode

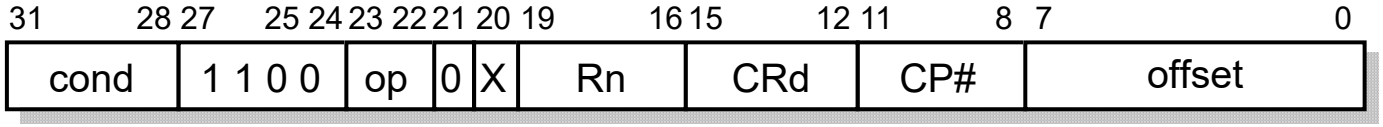


5.21 Unused instruction space

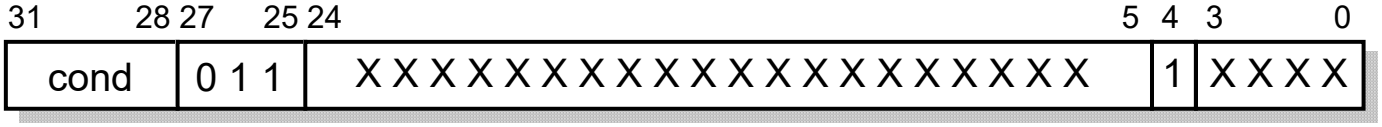
- Unused load/store instructions – These are likely to be used to support additional data transfer instructions, should these be required in the future



- Unused coprocessor instructions – is likely to be used to support any additional coprocessor instructions



- Undefined instruction space



5.22 Memory faults

- ARM processors allow the memory system to fault on any memory access
  - The memory system returns a signal that indicates that memory access has failed to complete correctly
  - The processor will then enter an exception handler and the system software will attempt to recover from the problem
- The most common sources of a memory fault in a general-purpose machine are
  - Page absent: the addressed memory location has been paged out to disk
  - Page protected: the addressed memory location is temporarily inaccessible
  - Soft memory errors: a soft error has been detected in the memory
    - memory has a hardware error detector but relies on software error correction

5.22 Memory faults

- Embedded systems
  - A hard disk is usually unavailable
  - The memory system is usually small
  - Many embedded systems will not use memory faults at all
- Memory faults
  - The ARM handles memory faults detected during instruction fetches (prefetch aborts) and those detected during data transfers (data aborts) separately

Exception	Mode	Vector address
Reset	SVC	0x00000000
Undefined instruction	UND	0x00000004
Software interrupt (SWI)	SVC	0x00000008
Prefetch abort (instruction fetch memory fault)	Abort	0x0000000C
Data abort (data access memory fault)	Abort	0x00000010
IRQ (normal interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

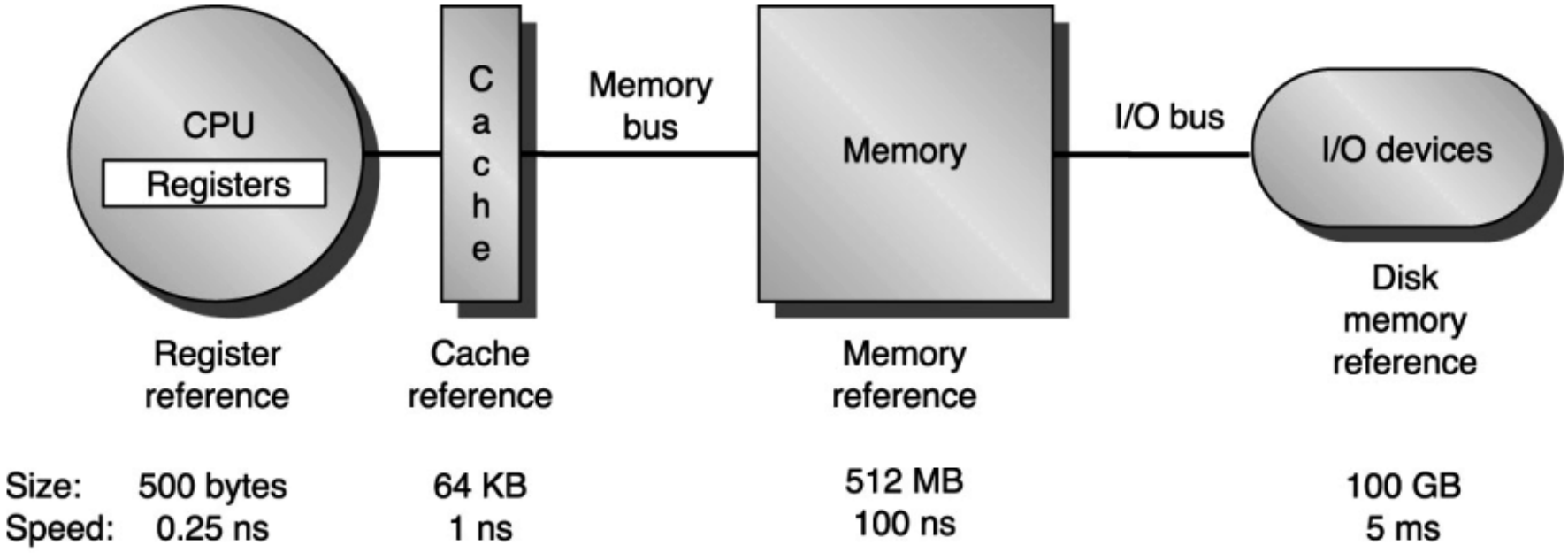


Fig. The levels in a typical memory hierarchy in embedded, desktop, and server computers

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	< 16 MB	< 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25–0.5	0.5–25	80–250	5,000,000
Bandwidth (MB/sec)	20,000–100,000	5000–10,000	1000–5000	20–150
Managed by	compiler	hardware	operating system	operating system/operator
Backed by	cache	main memory	disk	CD or tape

Fig. The typical levels in the hierarchy slow down and get larger as we move away from the CPU for a large workstation or small server

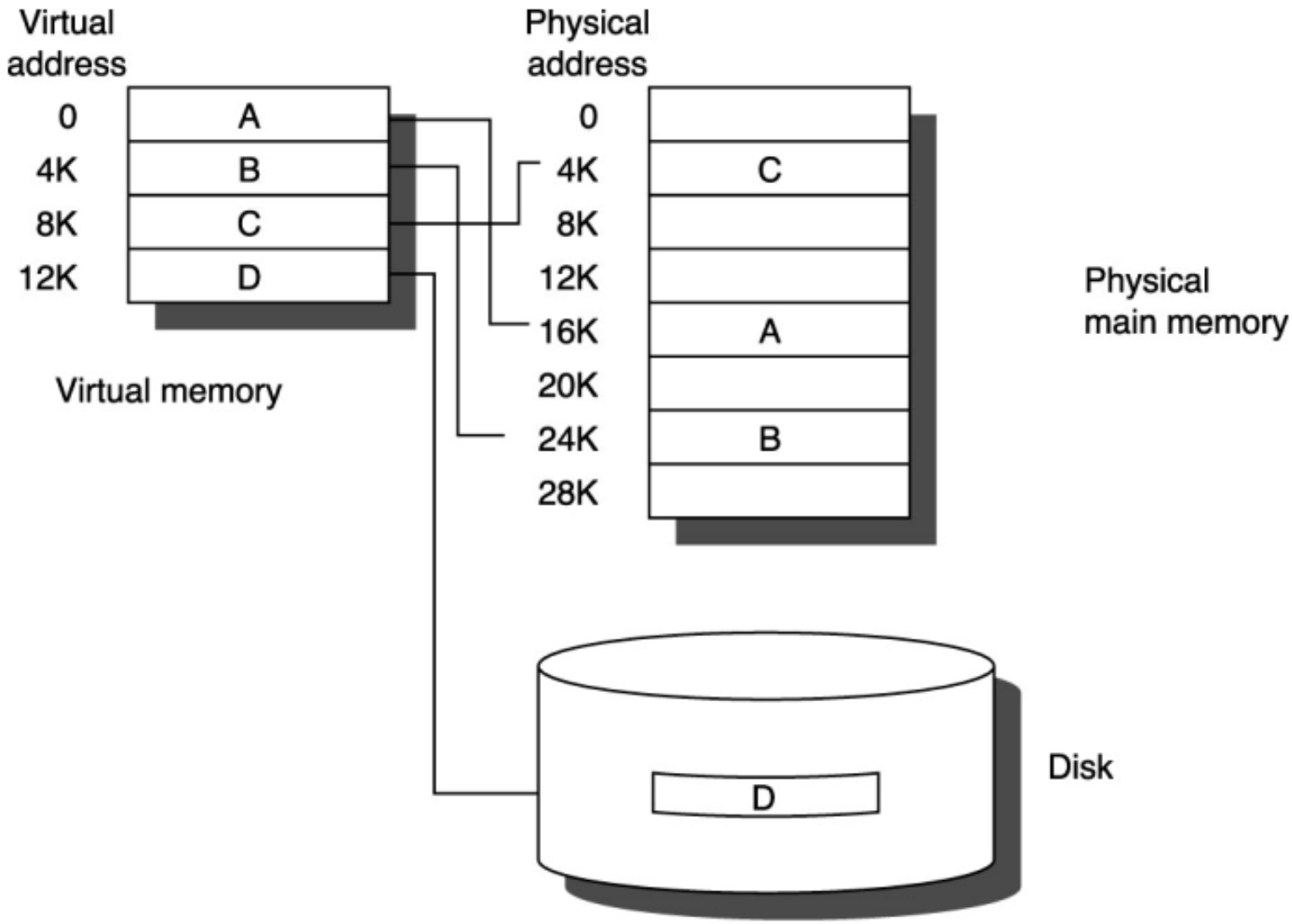


Fig. The logical program in its contiguous virtual address space in shown on the left.